Курс kiev-clrs – Лекция 9. Двоичные поисковые деревья

Олег Смирнов

31 мая 2009 г.

Содержание

1 Цель лекции

- Деревья как структуры данных
- Двоичные поисковые деревья (BST) и их связь с алгоритмом Quicksort
- Анализ высоты рандомизированного BST
- Алгоритмы работы с BST

2 Деревья

В теории графов, дерево – связный (ориентированный или неориентированный) граф, не содержащий циклов (для любой вершины есть один и только один способ добраться до любой другой вершины). Древовидная структура – тип организации, в котором каждый объект связан с хотя бы одним другим.

В программировании наиболее часто используются бинарные деревья, в которых число исходящих ребер не превосходит 2 и N-арные деревья с произвольным количеством исходящих ребер.

При хранении в памяти деревья обычно представляют в виде связной структуры, где каждый узел помимо ключа (key) хранит указатели на дочерние узлы и иногда на родительский. Для хранения N-арных деревьев используют структуру с левым дочерним и правым сестринским узлами (left-child, right-sibling representation). В этом случае вместо указателя на дочерние узлы каждый узел x хранит два указателя:

- в left_child[x] указатель на крайний левый дочерний узел узла x
- в right_sibling[x] хранится указатель на узел, расположенный на одном уровне с x справа от него

2.1 Свойства

1. Дерево не имеет кратных ребер и петель.
2. Любое дерево с n узлами содержит $n-1$ ребро. Более того, конечный связный граф является деревом тогда и только тогда, когда $B - P = 1$, здесь B – число узлов, P – число рёбер графа.

3. Граф является деревом тогда и только тогда, когда любые две различные его узла можно соединить единственным элементарным путём.

4. Любое дерево однозначно определяется расстояниями (длиной наименьшей цепи) между его концевыми (степени 1) узлами.

5. Любое дерево является двудольным графом. Любое дерево, содержащее счётное количество вершин, является планарным графом.

3 Двоичные (бинарные) поисковые деревья

Двоичным (бинарным) поисковым деревом называется бинарное дерево, для каждого узла x которого выполняется свойство:

- если узел y лежит в левом поддереве узла x, то $key[y] \leqslant key[x]$
- если узел y лежит в правом поддереве узла x, то $key[y] \geqslant key[x]$

"Хорошими" считаются сбалансированные бинарные деревья с высотой порядка $O(\lg n)$. У несбалансированных бинарных деревьев высота может достигать n. Время прохода по бинарному дереву пропорционально его высоте. Цель – построить бинарное дерево с высотой порядка $O(\lg n)$ в большинстве случаев. Один из способов – рандомизация, рассматривается в данной лекции.

3.1 Сортировка массива

Бинарные деревья поиска можно использовать для сортировки массива:

```
1  T ← ∅
2  for i ← 1 to n
3        do Tree_Insert(T, A[i])
4  Inorder_Tree_Walk(root[T])
```

Время работы алгоритма складывается из частей:

- $O(n)$ для обхода Inorder_Tree_Walk

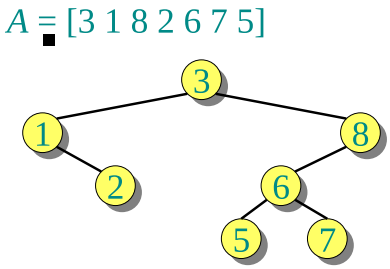

Рис. 1: Дерево работы BST sort

- $\Omega(n \lg n)$ для Tree_Insert в среднем и в лучшем случае (идеально сбалансированное бинарное дерево)
- $T = \sum_{x \in T} depth(x) = \Theta(n^2)$ для Tree_Insert в худшем случае (массиво уже отсортирован)

Поведение алгоритма похоже на поведение сортирвки Хоара – Quicksort.

4 BST и сортировка Хоара

Алгоритмы BST sort и Quicksort выполняют одинаковое количество сравнений, но в различном порядке.

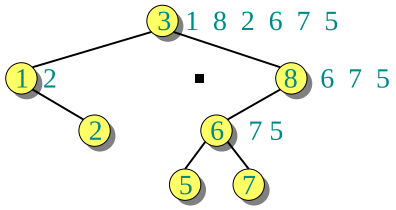

Рис. 2: Дерево работы Quicksort

Полученное дерево в точности совпадает с построенным в предыдущем примере.

Анализируя работу, можно увидеть, что алгоритм Quicksort в начале делает сравнение всех элементов с первым опорным (3), генерируя первое разбиение. BST sort также сравнивает каждый элемент в порядке добавления с корнем дерева (3). Аналогичные сравнения происходят для каждого элемента массива. Элемент, который в Quicksort становится опорным, в BST sort становится корнем поддерева.

4.1 Рандомизированный BST sort

1. Случайная перестановка элементов массива
2. Сортировка BST sort(A)

Время работы совпадает со временем рандомизированного Quicksort. Совпадает и мат. ожидание.

$$E[\text{time}] = E[\text{Randomized\_Quicksort}] = \Theta(n \lg n)$$

Нет смысла использовать BST sort для сортировки массива, т.к. время не отличается от Quicksort. Поиск по BST также не дает преимуществ по сравнению с бинарным поиском в просто отсортированном массиве. Полезность BST заключается в возможности добавлять элементы в структуру динамически, сохраняя ожидаемое время работы.

Ожидаемое время работы рандомизированного алгоритма BST sort $T(n)$ будет $\Theta(n \lg n)$. Время работы равно сумме глубины всех узлов дерева:

$$T(n) = \sum_x \text{depth} x$$

5 Анализ высоты BST

Интуитивно ясно, что ожидаемая высота дерева должна быть $\Theta(\lg n)$.

Ожидаемая *средняя* высота будет равна: $E[\frac{1}{n}\sum_{x \in T} \text{ depth } x] = \frac{\Theta(n \lg n)}{n} = \Theta(\lg n)$.

Ожидаемая средняя высота дерева $\Theta(\lg n)$ не означает, что высота всего дерева также будет $\Theta(\lg n)$. Например, если в дереве есть один из путей длинной $\sqrt{(}n) > \lg n$, а остальные пути $\lg n - \sqrt{(}n)$, средняя высота тем не менее будет равна:

$$\leqslant \frac{1}{n}(n \lg n + \sqrt{(}n)\sqrt{(}n)) = O(\lg n)$$

Теорема: $E[$высота рандомизированного BST$] = O(\lg n)$

Доказательство теоремы позволит показать, что в рандомизированном BST можно производить поиск за (ожидаемое) логарифмическое время.

Схема доказательства:

1. Неравенство Йенсена для выпуклой функции f: $f(E[X]) \leqslant E[f(X)]$
2. Вместо анализа X_n = случайная величина высоты BST анализ $Y_n = 2^{X_n}$
3. Доказательство $E[Y_n] = O(n^3)$
4. Поиск границы $E[2^{X_n}] = E[Y_n] = O(n^3)$
5. В соответствии с неравенством Йенсена $2^{E[X_n]} \leqslant E[2^{X_n}]$
6. После логарифмирования получим $E[X_n] \leqslant \lg O(n^3) = 3 \lg n + O(1)$

Определение: Функция $f\colon \mathbb{R} \to \mathbb{R}$ называется выпуклой для $x, y \in \mathbb{R}$ и $\forall \alpha, \beta \geqslant 0, \alpha + \beta = 1$, если $f(\alpha x + \beta y) \leqslant \alpha f(x) + \beta f(y)$.

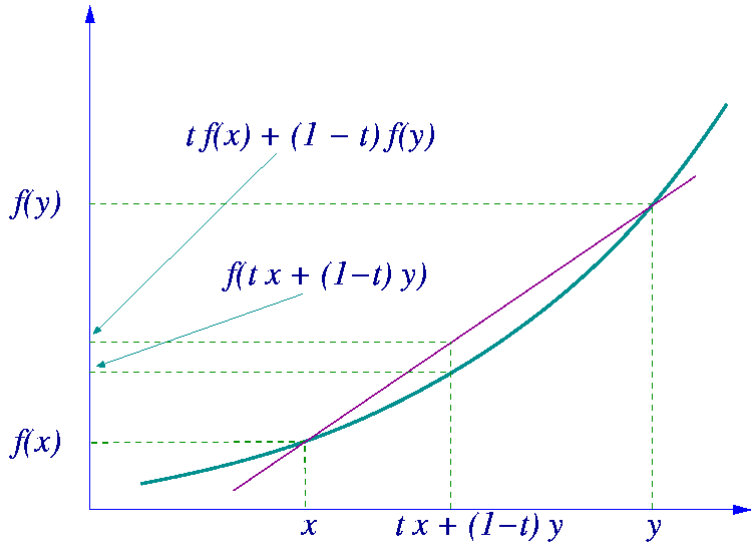

Рис. 3: Выпуклая функция

В доказетельстве неравенства Йенсена используется лемма: для выпуклой функции $f\colon \mathbb{R} \to \mathbb{R}$, $x_1, x_2, \ldots x_n \in \mathbb{R}$ и $\alpha_1, \alpha_2, \ldots \alpha_n \in \mathbb{R}$, $\sum_{k=1}^{n} \alpha_k = 1$, то $f(\sum_{k=1}^{n} \alpha_k x_k) \leqslant \sum_{k=1}^{n} \alpha_k f(x_k)$. Лемма аналогична определению, но для случая n переменных.

Графически видно, что линейная комбинация в левой части уравнения задаёт выпуклый многоугольник, все точки внутри которого лежат выше кривой.

Лемма доказывается по индукции:

$$f(\sum_{k=1}^{n} \alpha_k x_k) = \text{(по индукционной гипотезе)}$$

$$= f(a_n x_n + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{a_k}{1-\alpha_n} x_k) = \text{(отделяем меньшую часть)}$$

$$\text{т.к. } \alpha_n + (1-\alpha_n) = 1 \text{ и сумма всех } \alpha_k = 1 - \alpha_n$$

$$\leqslant \alpha_n f(x_n) + (1-\alpha_n) f(\sum_{k=1}^{n-1} \frac{a_k}{1-\alpha_n} x_k) \leqslant \text{(по свойству выпуклой функции)}$$

$$\leqslant \alpha_n f(x_n) + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{a_k}{1-\alpha_n} f(x_k) = \text{(по индукционной гипотезе)}$$

$$= \sum_{k=1}^{n} \alpha_k f(x_k) \text{(ч.т.д.)}$$

Случай для бесконечного числа переменных x_n доказывается аналогично: от обоих частей уравнения берется предел при $n \to \infty$.

Доказательство неравенства Йенсена: $f(E[X]) \leqslant E[f(X)]$, если f – выпуклая функция.

$$f(E[X]) = f(\sum_{x=-\infty}^{\infty} x Pr[X = x]) \leqslant \text{(т.к. сумма вероятностей равна 1)}$$

$$\leqslant \sum_{x=-\infty}^{\infty} Pr[X = x] f(x) =$$

$$= \sum_{y \in range(f)} y \sum_{x=f(x)=y} Pr[X = y] =$$

$$= \sum_{y \in range(f)} y Pr[f(X) = y] =$$

$$= E[f(X)] \text{(ч.т.д.)}$$

5.1 Ожидаемая высота

Пусть X_n – случайная величина высоты рандомизированного BST для n узлов, а $Y_n = 2_n^X$ – выпуклая функция.

Анализ высоты дерева похож на анализ алгоритма Quicksort в том смысле, что после выбора корня дерева r остальные элементы исходного массива разделяются на две части – меньшие r (пусть k элементов), которые попадут в левое поддерево и большие r $(n-k-1)$, которые попадут в правое поддерево. Каждое из поддеревьев также является случайным рандомизированным BST, что приводит к рекурсивному анализу алгоритма.

Если корень дерева r имеет ранг k, тогда $X_n = 1+\max(X_{k-1}, Xn-k)$, а $Y_n = 2max(Y_{k-1}, Y_{n-k})$.

Переход с помощью индикаторной случайной величины от совокупности случаев к сумме:

$$Z_{nk} = \begin{cases} 1, \text{если корень имеет ранг } k \\ 0, \text{иначе} \end{cases}$$

Очевидно $E[Z_{nk}] = 1/n$.

$$Y_n = \sum_{k=1}^{n} Z_{nk}(2max(Y_{k-1}, Y_{n-k}))$$

$$E[Y_n] = E[\sum_{k=1}^{n} Z_{nk}(2max(Y_{k-1}, Y_{n-k}))] =$$

(по свойству линейности)

$$= \sum_{k=1}^{n} E[Z_{nk}(2max(Y_{k-1}, Y_{n-k}))] =$$

(по независимости случайных величин)

$$= 2\sum_{k=1}^{n} E[Z_{nk}]E[max(Y_{k-1}, Y_{n-k})] =$$

(т.к. сумма больше максимума)

$$\leqslant \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}] =$$

(по свойству линейности)

$$= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

Решаем рекурентность методом подстановки, поставляя n^3:

$$\begin{aligned} E[Y_n] &\leqslant \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \leqslant \\ &\leqslant \frac{4}{n}\sum_{k=0}^{n-1} ck^3 \leqslant \\ &\leqslant \frac{4}{n}\int_0^n x^3 dx = \\ &= \frac{4c}{n}\frac{n^4}{4} = \\ &= cn^3 \end{aligned}$$

В итоге:

$$\begin{aligned} E[2^{X_n}] &= E[Y_n] = O(n^3) \\ 2^{E[X_n]} &\leqslant E[2^{X_n}] \\ E[X_n] &\leqslant \lg O(n^3) \\ &\text{(константа, присутствующая в O выносится)} \\ E[X_n] &\leqslant 3\lg n + O(1) \end{aligned}$$

6 Алгоритмы работы с BST

Пусть высота бинарного дерева равна h. Тогда алгоритм поиска элемента k в дереве с корнем x выполняется за время $O(h)$:

TREE_SEARCH(x, k)
1 **if** $x = NIL \mid k = key[x]$
2 **then return** x
3 **if** $k < key[x]$
4 **then return** $Tree\_Search(left[x], k)$
5 **else return** $Tree\_Search(right[x], k)$

Процедуру можно превратить в итеративную с помощью хвостовой рекурсии.

6.1 Алгоритмы вставки и удаления элемента

Алгоритм вставки узла z, у которого $key[z] = v$, $left[z] = NIL$, $right[z] = NIL$ в дерево T:

TREE_INSERT(T, z)

```
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z //Дерево T – пустое
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
```

Цикл в начале процедуры перемещает указатели вниз по дереву в зависимости от сравнения ключей $key[x]$ и $key[z]$, до тех пор, пока x не станет равным NIL. Это значение находится именно в той позиции, куда следует вставить узел z.

Процедура Tree_Delete рассматривает три возможных случая:

1. Если у узла z нет дочерних узлов, он просто удаляется из дерева
2. Если у узла один дочерний узел, он “склеивается” с родительским для z
3. Если дочерних узла два, то в дереве находим следующий за z узел y, у которого нет левого дочернего узла, убираем его из позиции, где он находился ранее и заменяем им узел z

Можно показать, что если у узла BST два дочерних узла, то у предшествующего узла нет правого дочернего, а у последующего – левого.

Если в дереве T существуют два узла a и b, $a < b$, такие, что $rank(a) = k$, $rank(b) = k+1$. Т.е. узел b является последующим за a. Левым дочерним узлом узла b может являтся только элемент $x < b$. Но из условия

предшествия следует, что $x < a$ и $rank(x) < rank(a)$. Т.к. узлы a и b уже размещены в дереве, алгоритм Tree_Insert поместит x в левое поддерево элемента a, но не b.

Процедура Tree_Successor возвращает следующий элемент за аргументом в отсортированной последовательности.

```
TREE_DELETE(T, z)
 1  if left[z] = NIL | right[z] = NIL
 2     then y ← z
 3     else  y ← Tree_Successor(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else  x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else  if y = left[p[y]]
12              then left[p[y]] ← x
13              else  right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          //Копирование сопутствующих данных в z
17  return y
```

Очевидно, что балансировку дерева можно нарушить, вставляя или удаляя специально подобранные элементы. Для борьбы с таким поведением существуют специальные структуры данных и соответствующие алгоритмы: красно-чёрные деревья и AVL-деревья.